Defining Fields

This section describes how you define the fields (database fields and user-defined) you wish to use in a program. These fields can be database fields and user-defined fields. It contains information that applies to all fields in general and to user-defined fields in particular. The particulars of database fields are described in Database Access.

- DEFINE DATA Statement
- Structure of a DEFINE DATA Statement - Level Numbers
- User-Defined Variables
- User-Defined Constants
- Initial Values (and the RESET Statement)
- Redefining Fields
- Arrays
- Data Blocks

Please note that only the major options of the DEFINE DATA statement are discussed here. Further options are described in the Natural Statements documentation.

DEFINE DATA Statement

The first statement in a Natural program must always be a DEFINE DATA statement. In this statement, you define all the fields - database fields as well as user-defined variables - that are to be used in the program.

All fields to be used *must be* defined in the DEFINE DATA statement.

There are two ways to define the fields:

- The fields can be defined within the DEFINE DATA statement itself.
- The fields can be defined outside the program in a local or global data area, with the DEFINE DATA statement referencing that data area.

If fields are used by multiple programs/routines, they should be defined in a data area outside the programs.

For a clear application structure, it is usually better to define fields in data areas outside the programs.

Data areas are created and maintained with the data area editor, which is described in your Natural User’s Guide.

In the first example below, the fields are defined within the DEFINE DATA statement of the program. In the second example, the same fields are defined in a local data area, and the DEFINE DATA statement only contains a reference to that data area.

Example 1 - Fields Defined within a DEFINE DATA Statement:

```
DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 NAME
    2 FIRST-NAME
    2 PERSONNEL-ID
  1 #VARI-A (A20)
  1 #VARI-B (N3.2)
  1 #VARI-C (I4)
  END-DEFINE
  ...
```

Example 2 - Fields Defined in a Separate Data Area:

Program:

```
DEFINE DATA LOCAL
    USING LDA39
  END-DEFINE
  ...
```

Local Data Area "LDA39":

```
  I T L Name                            F Leng Index/Init/EM/Name/Comment
  - - - ------------------------------- - ---- --------------------------------
    V 1 VIEWEMP                                EMPLOYEES
      2 NAME                            A   20
      2 FIRST-NAME                      A   20
      2 PERSONNEL-ID                    A    8
      1 #VARI-A                         A   20
      1 #VARI-B                         N  3.2
      1 #VARI-C                         I    4
```


Structure of a DEFINE DATA Statement - Level Numbers

Level numbers are used within the DEFINE DATA statement to indicate the structure and grouping of the definitions. This is relevant with:

- view definitions
- field groups
- redefinitions

Level numbers are 1- or 2-digit numbers in the range from 01 to 99 (the leading "0" is optional).

Generally, variable definitions are on level 1.

The level numbering in view definitions, redefinitions and groups must be sequential; no level numbers may be skipped.

Level Numbers in View Definitions

If you define a view, the specification of the view name must be on level 1, and the fields the view is comprised of must be on level 2. (For details on view definitions, see Database Access.)

Example of Level Numbers in View Definition:

```
DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 NAME
    2 FIRST-NAME
    2 BIRTH
  ...
  END-DEFINE
```

Level Numbers in Field Groups

The definition of groups provides a convenient way of referencing a series of consecutive fields. If you define several fields under a common group name, you can reference the fields later in the program by specifying only the group name instead of the names of the individual fields.

The group name must be specified on level 1, and the fields contained in the group must be one level lower.

For group names, the same naming conventions apply as for user-defined variables.

Example of Level Numbers in Group:

```
DEFINE DATA LOCAL
  1 #FIELDA (N2.2)
  1 #FIELDB (I4)
  1 #GROUPA
    2 #FIELDC (A20)
    2 #FIELDD (A10)
    2 #FIELDE (N3.2)
  1 #FIELDF (A2)
  ...
  END-DEFINE
```

In this example, the fields #FIELDC, #FIELDD and #FIELDE are defined under the common group name #GROUPA. The other three fields are not part of the group. Note that #GROUPA only serves as a group name and is not a field in its own right (and therefore does not have a format/length definition).

Level Numbers in Redefinitions

If you redefine a field, the REDEFINE option must be on the same level as the original field, and the fields resulting from the redefinition must be one level lower. (For details on redefinitions, see the section Redefining Fields.)

Example of Level Numbers in Redefinition:

```
DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF STAFFDDM
    2 BIRTH
    2 REDEFINE BIRTH
      3 #YEAR-OF-BIRTH  (N4)
      3 #MONTH-OF-BIRTH (N2)
      3 #DAY-OF-BIRTH   (N2)
  1 #FIELDA (A20)
  1 REDEFINE #FIELDA
    2 #SUBFIELD1 (N5)
    2 #SUBFIELD2 (A10)
    2 #SUBFIELD3 (N5)
  ...
  END-DEFINE
```

In this example, the database field BIRTH is redefined as three user-defined variables, and the user-defined variable #FIELDA is redefined as three other user-defined variables.

User-Defined Variables

User-defined variables are fields which you define yourself in a program. They are used to store values or intermediate results obtained at some point in program processing for additional processing or display.

You define a user-defined variable by specifying its name and its format/length in the DEFINE DATA statement.

Example:

In this example, a user-defined variable of alphanumeric format and a length of 10 positions is defined with the name #FIELD1.

```
DEFINE DATA LOCAL
  1 #FIELD1 (A10)
  ...
  END-DEFINE
```

The following topics are covered below:

- Names of User-Defined Variables
- Format and Length of User-Defined Variables

Names of User-Defined Variables

The name of a user-defined variable can be 1 to 32 characters long.

Note:
You may use variable names of over 32 characters (for example, in complex applications where longer meaningful variable names enhance the readability of programs); however, only the first 32 characters are significant and must therefore be unique, the remaining characters will be ignored by Natural.

The name of a user-defined variable must not be a Natural reserved word.

Within one Natural program, you should not use the same name for a user-defined variable and a database field.

The name of a user-defined variable can consist of the following characters:

Character	Explanation
A to **Z**	alphabetical characters
0 to **9**	numeric characters
-	hyphen
@	at sign
_	underline
/	slash
$	dollar sign
§	paragraph sign
&	ampersand
#	hash / number sign
+	plus sign (only allowed as first character)

The first character of the name must be one of the following:

- an upper-case alphabetical character
- **#**
- **+**
- **&**

Note:
In this section, the names of all user-defined variables begin with a hash sign (#); this avoids any naming conflicts with database fields or Natural reserved words.

If the first character is a "#", "+", or "&", the name must consist of at least one additional character.

"+" as the first character of a name is only allowed for application-independent variables (AIVs) and variables in a global data area. Names of AIVs **must** begin with a "+".

"&" as the first character of a name is used in conjunction with dynamic source program modification (see the RUN statement in the Natural Statements documentation) and when defining processing rules (see the map editor description in your Natural User's Guide).

Format and Length of User-Defined Variables

Format and length of a user-defined variable are specified in parentheses after the variable name.

A user-defined variable can have one of the following formats:

A	Alphanumeric
B	Binary
C	Attribute Control
D	Date
F	Floating Point
I	Integer
L	Logical
N	Numeric unpacked
P	Packed numeric
T	Time

Information on possible lengths of user-defined variables is provided in the Natural Reference documentation,

Examples of User-Defined Variables:

```
DEFINE DATA LOCAL
  #A1 (A10)       /* Alphanumeric, 10 positions.
  #A2 (B4)        /* Binary, 4 positions.
  #A3 (P4)        /* Packed numeric, 4 positions and 1 sign position.
  #A4 (N7.2)      /* Unpacked numeric,
                  /* 7 positions before and 2 after decimal point.
  #A5 (N7.)       /* Invalid definition!!!
  #A6 (P7.2)      /* Packed numeric, 7 positions before and 2 after decimal point
                  /* and 1 sign position.
  #INT1 (I1)      /* Integer, 1 byte.
  #INT2 (I2)      /* Integer, 2 bytes.
  #INT3 (I3)      /* Invalid definition!!!
  #INT4 (I4)      /* Integer, 4 bytes.
  #INT5 (I5)      /* Invalid definition!!!
  #FLT4 (F4)      /* Floating point, 4 bytes.
  #FLT8 (F8)      /* Floating point, 8 bytes.
  #FLT2 (F2)      /* Invalid definition!!!
  #DATE (D)       /* Date (internal format/length P6).
  #TIME (T)       /* Time (internal format/length P12).
  #SWITCH (L)     /* Logical, 1 byte (TRUE or FALSE).
  ...
  END-DEFINE
```

Note:
When a user-defined variable of format P is output with a DISPLAY, WRITE, or INPUT statement, Natural internally converts the format to N for the output.

User-Defined Constants

Constants can be used throughout Natural programs. This section discusses the types of constants that are supported and how they are used:

- Numeric Constants
- Alphanumeric Constants
- Date and Time Constants
- Hexadecimal Constants
- Logical Constants
- Floating Point Constants
- Attribute Constants
- Defining Named Constants

Numeric Constants

A numeric constant may contain 1 to 29 numeric digits.

A numeric constant used with a COMPUTE, MOVE, or arithmetic statement may contain a decimal point and sign notation.

Examples:

```
MOVE 3 TO #XYZ
  COMPUTE #PRICE = 23.34
  COMPUTE #XYZ = -103
  COMPUTE #A = #B * 6074
```

Alphanumeric Constants

An alphanumeric constant may contain 1 to 253 alphanumeric characters.

An alphanumeric constant must be enclosed in either apostrophes (’) or quotation marks (").

Examples:

```
MOVE ’ABC’ TO #XYZ
  MOVE ’% INCREASE’ TO #TITLE
  DISPLAY "LAST-NAME" NAME
```

If you want an apostrophe to be part of an alphanumeric constant that is enclosed in apostrophes, you must write this as two apostrophes or as a single quotation mark.

If you want an apostrophe to be part of an alphanumeric constant that is enclosed in quotation marks, you write this as a single apostrophe.

Example:

If you want the following to be output:

```
HE SAID, 'HELLO'
```

you can use any of the following notations:

```
WRITE 'HE SAID, ''HELLO'''
  WRITE 'HE SAID, "HELLO"'
  WRITE "HE SAID, ""HELLO"""
  WRITE "HE SAID, 'HELLO'"
```

An alphanumeric constant that is used to assign a value to a user-defined variable must not be split between statement lines.

Alphanumeric constants may be concatenated to form a single value by use of a hyphen.

Examples:

```
MOVE 'XXXXXX' -
       'YYYYYY' TO #FIELD

  MOVE "ABC" - 'DEF' TO #FIELD
```

In this way, alphanumeric constants can also be concatenated with hexadecimal constants.

Date and Time Constants

A date constant may be used in conjunction with a format D variable. Date constants may have the following formats:

D'*yyyy-mm-dd*'	International date format
D'*dd.mm.yyyy*'	German date format
D'*dd/mm/yyyy*'	European date format
D'*mm/dd/yyyy*'	USA date format

where *dd* represent the number of the day, *mm* the number of the month and *yyyy* the year.

Example:

```
DEFINE DATA LOCAL
  1 #DATE (D)
  END-DEFINE
  ...
  MOVE D'1997-08-11' TO #DATE
  ...
```

The default date format is controlled by the profile parameter DTFORM as set by the Natural administrator.

A time constant may be used in conjunction with a format T variable. A time constant has the following format:

T'*hh:ii:ss***'**

where *hh* represents hours,*ii* minutes and *ss* seconds.

Example:

```
DEFINE DATA LOCAL
  1 #TIME (T)
  END-DEFINE
  ...
  MOVE T'11:33:00' TO #TIME
```


Hexadecimal Constants

A hexadecimal constant may be used to enter a value which cannot be entered as a standard keyboard character.

A hexadecimal constant is prefixed with an "H". The constant itself must be enclosed in apostrophes and may consist of the hexadecimal characters 0 - 9, A - F. Two hexadecimal characters are required to represent one byte of data.

The hexadecimal representation of a character varies, depending on whether your computer uses an ASCII or EBCDIC character set. Wenn you transfer hexadecimal constants to another computer, you may therefore have to convert the characters.

ASCII Examples:

```
H’313233’      (equivalent to the alphanumeric constant ’123’)
  H’414243’      (equivalent to the alphanumeric constant ’ABC’)
```

EBCDIC Examples:

```
H’F1F2F3’      (equivalent to the alphanumeric constant ’123’)
  H’C1C2C3’      (equivalent to the alphanumeric constant ’ABC’)
```

Hexadecimal constants may be concatenated by using a hyphen between the constants.

ASCII Example:

```
H’414243’ - H’444546’   (equivalent to ’ABCDEF’)
```

EBCDIC Example:

```
H’C1C2C3’ - H’C4C5C6’   (equivalent to ’ABCDEF’)
```

Logical Constants

The logical constants "TRUE" and "FALSE" may be used to assign a logical value to a field defined with format L.

Example:

```
DEFINE DATA LOCAL
  1 #FLAG (L)
  END-DEFINE
  ...
  MOVE TRUE TO #FLAG
  ...
  IF #FLAG ...
     statement ...
     MOVE FALSE TO #FLAG
  END-IF
  ...
```

Floating Point Constants

Floating point constants can be used with variables defined with format F.

Example:


```
DEFINE DATA LOCAL
  1 #FLT1 (F4)
  END-DEFINE
  ...
  COMPUTE #FLT1 = -5.34E+2
  ...
```

Attribute Constants

Attribute constants can be used with variables defined with format C (control variables). This type of constant must be enclosed within parentheses.

The following attributes may be used:

AD=D	default	**CD=BL**	blue
AD=B	blinking	**CD=GR**	green
AD=I	intensified	**CD=NE**	neutral
AD=N	non-display	**CD=PI**	pink
AD=V	reverse video	**CD=RE**	red
AD=U	underlined	**CD=TU**	turquoise
AD=C	cursive/italic	**CD=YE**	yellow
AD=Y	dynamic attribute		
AD=P	protected		

Example:

```
DEFINE DATA LOCAL
  1 #ATTR  (C)
  1 #FIELD (A10)
  END-DEFINE
  ...
  MOVE (AD=I CD=BL) TO #ATTR
  ...
  INPUT #FIELD (CV=#ATTR)
  ...
```


Defining Named Constants

If you need to use the same constant value several times in a program, you can reduce the maintenance effort by defining a named constant: you define a field in the DEFINE DATA statement, assign a constant value to it, and use the field name in the program instead of the constant value. Thus, when the value has to be changed, you only have to change it once in the DEFINE DATA statement and not everywhere in the program where it occurs.

You specify the constant value in angle brackets with the keyword "CONSTANT" after the field definition in the DEFINE DATA statement. If the value is alphanumeric, it must be enclosed in apostrophes.

Example:

```
DEFINE DATA LOCAL
  1 #FIELDA (N3) CONSTANT <100>
  1 #FIELDB (A5) CONSTANT <'ABCDE'>
  END-DEFINE
  ...
```

During the execution of the program, the value of such a named constant cannot be modified.

Initial Values

You can assign an initial value to a user-defined variable. You specify the initial value in angle brackets with the keyword "INIT" after the variable definition in the DEFINE DATA statement. If the initial value is alphanumeric, it must be enclosed in apostrophes.

Example:

```
DEFINE DATA LOCAL
  1 #FIELDA (N3)  INIT <100>
  1 #FIELDB (A20) INIT <'ABC'>
  END-DEFINE
  ...
```

The initial value for a field may also be the value of a Natural system variable.

Example of System Variable as Initial Value:

```
DEFINE DATA LOCAL
  1 #MYDATE (D) INIT <*DATX>
  END-DEFINE
  ...
```

As initial value, a variable can also be filled, entirely or partially, with a specific single character or string of characters (only possible for alphanumeric variables).

With the option **FULL LENGTH**<*character(s)*> the entire field is filled with the specified *character(s)*.

With the option **LENGTH***n* <*character(s)*> the first n positions of the field are filled with the specified *character(s)*.

Example of FULL LENGTH:

In this example, the entire field will be filled with asterisks.

```
DEFINE DATA LOCAL
  1 #FIELD (A25) INIT FULL LENGTH <'*'>
  END-DEFINE
  ...
```

Example of LENGTH n:

In this example, the first 4 positions of the field will be filled with exclamation marks.

```
DEFINE DATA LOCAL
  1 #FIELD (A25) INIT LENGTH 4 <'!'>
  END-DEFINE
  ...
```


Default Initial Values

If you specify no initial value for a field, the field will be initialised with a default initial value (null value) depending on its format:

Format	Default Initial Value
B, F, I, N, P	0
A	blank
L	F(ALSE)
D	D’ ’
T	T’00:00:00’
C	(AD=D)

The RESET Statement

The RESET statement is used to set the value of a field to a null value, or to a specific initial value.

- RESET (without INITIAL) sets the value of each specified field to a null value.
- RESET INITIAL sets each specified field to the initial value as defined for the field in the DEFINE DATA statement.

Example:

```
DEFINE DATA LOCAL
  1 #FIELDA (N3)  INIT <100>
  1 #FIELDB (A20) INIT <’ABC’>
  1 #FIELDC (I4)  INIT <5>
  END-DEFINE
  ...
  ...
  RESET #FIELDA                          /* resets field value to null
  ...
  RESET INITIAL #FIELDA #FIELDB #FIELDC /* resets field values to initial values
  ...
```

Redefining Fields

Redefinition is used to change the format of a field, or to divide a single field into segments.

The REDEFINE option of the DEFINE DATA statement can be used to redefine a single field - either a user-defined variable or a database field - as one or more new fields. A group can also be redefined.

Important:
Dynamic variables are not allowed.

The REDEFINE option redefines byte positions of a field from left to right, regardless of the format. Byte positions must match between original field and redefined field(s).

The redefinition must be specified immediately after the definition of the original field.

In the following example, the database field BIRTH is redefined as three new user-defined variables:

```
DEFINE DATA LOCAL
  01 EMPLOY-VIEW VIEW OF STAFFDDM
     02 NAME
     02 BIRTH
     02 REDEFINE BIRTH
        03 #BIRTH-YEAR  (N4)
        03 #BIRTH-MONTH (N2)
        03 #BIRTH-DAY   (N2)
  END-DEFINE
  ...
```

In the following example, the group #VAR2, which consists of two user-defined variables of format N and P respectively, is redefined as a variable of format A:

```
DEFINE DATA LOCAL
  01 #VAR1 (A15)
  01 #VAR2
     02 #VAR2A (N4.1)
     02 #VAR2B (P6.2)
  01 REDEFINE #VAR2
     02 #VAR2RD (A10)
  END-DEFINE
  ...
```

With the notation **FILLER** *n***X** you can define *n* filler bytes - that is, segments which are not to be used - in the field that is being redefined. (The definition of trailing filler bytes is optional.)

In the following example, the user-defined variable #FIELD is redefined as three new user-defined variables, each of format/length A2. The FILLER notations indicate that the 3rd and 4th and 7th to 10th bytes of the original field are not be used.

```
DEFINE DATA LOCAL
  1 #FIELD (A12)
  1 REDEFINE #FIELD
    2 #RFIELD1 (A2)
    2 FILLER 2X
    2 #RFIELD2 (A2)
    2 FILLER 4X
    2 #RFIELD3 (A2)
  END-DEFINE
  ...
```

The following program illustrates the use of a redefinition:

```
** Example Program 'DDATAX01'
  DEFINE DATA LOCAL
  01 VIEWEMP VIEW OF EMPLOYEES
     02 NAME
     02 FIRST-NAME
     02 SALARY  (1:1)
  01 #PAY       (N9)
  01 REDEFINE #PAY
     02 FILLER 3X
     02 #USD    (N3)
     02 #000    (N3)
  END-DEFINE
  *
  READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
    MOVE SALARY (1) TO #PAY
    DISPLAY NAME FIRST-NAME #PAY #USD #000
  END-READ
  END
```

Note how #PAY and the fields resulting from its definition are displayed:

```
Page      1                                                99-08-08  17:48:59

          NAME              FIRST-NAME          #PAY    #USD #000
  ------------------- ------------------- ---------- ---- ----

  JONES               VIRGINIA                 46000   46    0
  JONES               MARSHA                   50000   50    0
  JONES               ROBERT                   31000   31    0
```

Arrays

Natural supports the processing of *arrays*. Arrays are multi-dimensional tables, that is, two or more logically related elements identified under a single name. Arrays can consist of single data elements of multiple dimensions or hierarchical data structures which contain repetitive structures or individual elements. In Natural, an array can be one-, two- or three-dimensional. It can be an independent variable, part of a larger data structure or part of a database view.

The following topics are covered below:

- Defining Arrays
- Initial Values for Arrays
- Assigning Initial Values to One-Dimensional Arrays
- Assigning Initial Values to Two-Dimensional Arrays
- A Three-Dimensional Array
- Arrays as Part of a Larger Data Structure
- Database Arrays
- Using Arithmetic Expressions in Index Notation
- Arithmetic Support for Arrays

Defining Arrays

To define an array variable, after the format and length you specify a slash followed by a so-called *index notation*, that is, the number of occurrences of the array.

Important:
Dynamic variables are not allowed.

For example, the following array has three occurrences, each occurrence being of format/length A10:

```
DEFINE DATA LOCAL
  1 #ARRAY (A10/1:3)
  END-DEFINE
  ...
```

To define a two-dimensional array, you specify an index notation for both dimensions:

```
DEFINE DATA LOCAL
  1 #ARRAY (A10/1:3,1:4)
  END-DEFINE
  ...
```

A two-dimensional array can be visualized as a table. The array defined in the example above would be a table that consists of 3 "rows" and 4 "columns":

Initial Values for Arrays

To assign initial values to one or more occurrences of an array, you use an INIT specification, similar to that for "ordinary" variables.

Assigning Initial Values to One-Dimensional Arrays

The following examples illustrate how initial values are assigned to a one-dimensional array.

- To assign an initial value to one occurrence, you specify:

```
1 #ARRAY (A1/1:3) INIT (2) <'A'>
```

"A" is assigned to the second occurrence.

- To assign the same initial value to all occurrences, you specify:

```
1 #ARRAY (A1/1:3) INIT ALL <'A'>
```

"A" is assigned to every occurrence. Alternatively, you could specify:

```
1 #ARRAY (A1/1:3) INIT (*) <'A'>
```

- To assign the same initial value to a range of several occurrences, you specify:

```
1 #ARRAY (A1/1:3) INIT (2:3) <'A'>
```

"A" is assigned to the second to third occurrence.

- To assign a different initial value to every occurrence, you specify:

```
1 #ARRAY (A1/1:3) INIT <'A','B','C'>
```

"A" is assigned to the first occurrence, "B" to the second, and "C" to the third.

- To assign different initial values to some (but not all) occurrences, you specify:

```
1 #ARRAY (A1/1:3) INIT (1) <'A'> (3) <'C'>
```

"A" is assigned to the first occurrence, and "C" to the third; no value is assigned to the second occurrence.

Alternatively, you could specify:

```
1 #ARRAY (A1/1:3) INIT <'A',,'C'>
```

- If fewer initial values are specified than there are occurrences, the last occurrences remain empty:

```
1 #ARRAY (A1/1:3) INIT <'A','B'>
```

"A" is assigned to the first occurrence, and "B" to the second; no value is assigned to the third occurrence.

Assigning Initial Values to Two-Dimensional Arrays

The following examples illustrate how initial values are assigned to a two-dimensional array.

For the examples, let us assume a two-dimensional array with three occurrences in the first dimension ("rows") and four occurrences in the second dimension ("columns"):

```
1 #ARRAY (A1/1:3,1:4)
```

Vertical: First Dimension (1:3), Horizontal: Second Dimension (1:4):

(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)

The first set of examples illustrates how the *same* initial value is assigned to occurrences of a two-dimensional array; the second set of examples illustrates how *different* initial values are assigned.

In the examples, please note in particular the usage of the notations "*" and "V". Both notations refer to *all* occurrences of the dimension concerned: "*" indicates that all occurrences in that dimension are initialized with the *same* value, while "V" indicates that all occurrences in that dimension are initialized with *different* values.

- Assigning the Same Value
- Assigning Different Values

Assigning the Same Value

- To assign an initial value to one occurrence, you specify:

		A	

To assign the same initial value to one occurrence in the second dimension - in all occurrences of the first dimension - you specify:

```
1 #ARRAY (A1/1:3,1:4) INIT (*,3) <'A'>
```

		A	
		A	
		A	

- To assign the same initial value to a range of occurrences in the first dimension - in all occurrences of the second dimension - you specify:

```
1 #ARRAY (A1/1:3,1:4) INIT (2:3,*) <'A'>
```

A	A	A	A
A	A	A	A

- To assign the same initial value to a range of occurrences in each dimension, you specify:

```
1 #ARRAY (A1/1:3,1:4) INIT (2:3,1:2) <'A'>
```

A	A		
A	A		

- To assign the same initial value to all occurrences (in both dimensions), you specify:

```
1 #ARRAY (A1/1:3,1:4) INIT ALL <'A'>
```

A	A	A	A
A	A	A	A
A	A	A	A

Alternatively, you could specify:

```
1 #ARRAY (A1/1:3,1:4) INIT (*,*) <'A'>
```

Assigning Different Values

```
1 #ARRAY (A1/1:3,1:4) INIT (V,2) <'A','B','C'>
```

	A		
	B		
	C		

```
1 #ARRAY (A1/1:3,1:4) INIT (V,2:3) <'A','B','C'>
```

	A	A	
	B	B	
	C	C	

```
1 #ARRAY (A1/1:3,1:4) INIT (V,*) <'A','B','C'>
```

A	A	A	A
B	B	B	B
C	C	C	C

```
1 #ARRAY (A1/1:3,1:4) INIT (V,*) <'A',,'C'>
```

A	A	A	A
C	C	C	C

```
1 #ARRAY (A1/1:3,1:4) INIT (V,*) <'A','B'>
```

A	A	A	A
B	B	B	B

```
1 #ARRAY (A1/1:3,1:4) INIT (V,1) <'A','B','C'>
                      (V,3) <'D','E','F'>
```

A		D	
B		E	
C		F	


```
1 #ARRAY (A1/1:3,1:4) INIT (3,V) <'A','B','C','D'>
```

A	B	C	D

```
1 #ARRAY (A1/1:3,1:4) INIT (*,V) <'A','B','C','D'>
```

A	B	C	D
A	B	C	D
A	B	C	D

```
1 #ARRAY (A1/1:3,1:4) INIT (2,1) <'A'> (*,2) <'B'>
                          (3,3) <'C'> (3,4) <'D'>
```

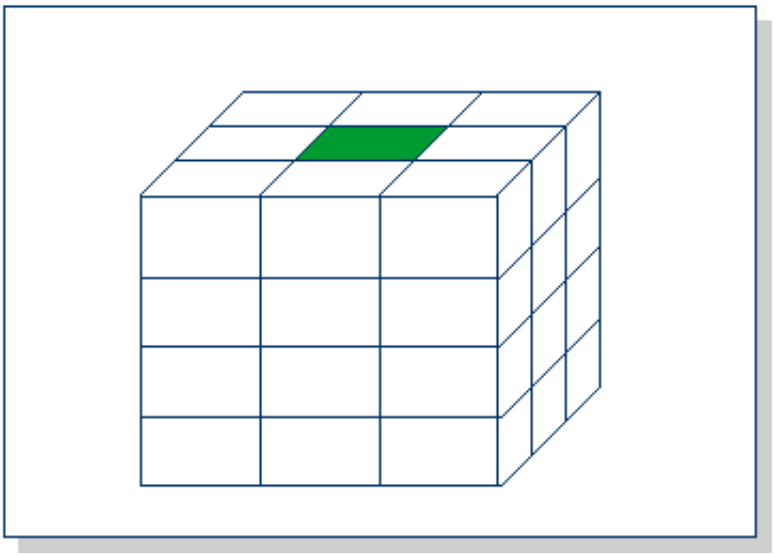
	B		
A	B		
	B	C	D

```
1 #ARRAY (A1/1:3,1:4) INIT (2,1) <'A'> (V,2) <'B',C',D'>
                          (3,3) <'E'> (3,4) <'F'>
```

	B		
A	C		
	D	E	F

A Three-Dimensional Array

A three-dimensional array could be visualized as follows:



The array illustrated here would be defined as follows (at the same time assigning an initial value to the highlighted field in row 1, column 2, plane 2):

```
DEFINE DATA LOCAL
  1 #ARRAY2
    2 #ROW (1:4)
      3 #COLUMN (1:3)
        4 #PLANE (1:3)
          5 #FIELD2 (P3) INIT (1,2,2) <100>
  END-DEFINE
  ...
```

If defined as a local data area in the data area editor, the same array would look as follows:

```
  I T L Name                            F Leng Index/Init/EM/Name/Comment
  - - - ------------------------------- - ---- --------------------------------
      1 #ARRAY2
      2 #ROW                                   (1:4)
      3 #COLUMN                                (1:3)
      4 #PLANE                                 (1:3)
  I   5 #FIELD2                         P    3
```


Arrays as Part of a Larger Data Structure

The multiple dimensions of an array make it possible to define data structures analogous to COBOL or PL1 structures.

Example:

```
DEFINE DATA LOCAL
  1 #AREA
    2 #FIELD1 (A10)
    2 #GROUP1 (1:10)
      3 #FIELD2 (P2)
      3 #FIELD3 (N1/1:4)
  END-DEFINE
  ...
```

In this example, the data area #AREA has a total size of:

10 + (10 * (2 + (1 * 4))) bytes = 70 bytes.

#FIELD1 is alphanumeric and 10 bytes long. #GROUP1 is the name of a sub-area within #AREA which consists of 2 fields and has 10 occurrences. #FIELD2 is packed numeric, length 2. #FIELD3 is the second field of #GROUP1 with four occurrences, and is numeric, length 1.

To reference a particular occurrence of #FIELD3, two indices are required: first, the occurrence of #GROUP1 must be specified, and second, the particular occurrence of #FIELD3 must also be specified. For example, in an ADD statement later in the same program, #FIELD3 would be referenced as follows:

```
ADD 2 TO #FIELD3 (3,2)
```


Database Arrays

Adabas supports array structures within the database in the form of *multiple-value fields* and *periodic groups*. These are described in Database Access.

The following example shows a DEFINE DATA view containing a multiple-value field:

```
DEFINE DATA LOCAL
    1 EMPLOYEES-VIEW VIEW OF EMPLOYEES
      2 NAME
      2 ADDRESS-LINE (1:10) /* <--- MULTIPLE-VALUE FIELD
  END-DEFINE
  ...
```

The same view in a local data area would look as follows:

```
  I T L Name                           F Leng Index/Init/EM/Name/Comment
  - - - ------------------------------ - ---- --------------------------------
    V 1 EMPLOYEES-VIEW                        EMPLOYEES
      2 NAME                           A   20
    M 2 ADDRESS-LINE                   A   20 (1:10) /* MU-FIELD
```

Using Arithmetic Expressions in Index Notation

A simple arithmetic expression may also be used to express a range of occurrences in an array.

Examples:

MA (I:I+5) Values of the field MA are referenced, beginning with value I and ending with value I+5.

MA (I+2:J-3) Values of the field MA are referenced, beginning with value I+2 and ending with value J-3.

Only the arithmetic operators "+" and "-" may be used in index expressions.

Arithmetic Support for Arrays

Arithmetic support for arrays include operations at array level, at row/column level, and at individual element level. Only simple arithmetic expressions are permitted with array variables, with only one or two operands and an optional third variable as the receiving field. Only the arithmetic operators "+" and "-" are allowed for expressions defining index ranges.

Examples of Array Arithmetics:

The following examples assume the following field definitions:

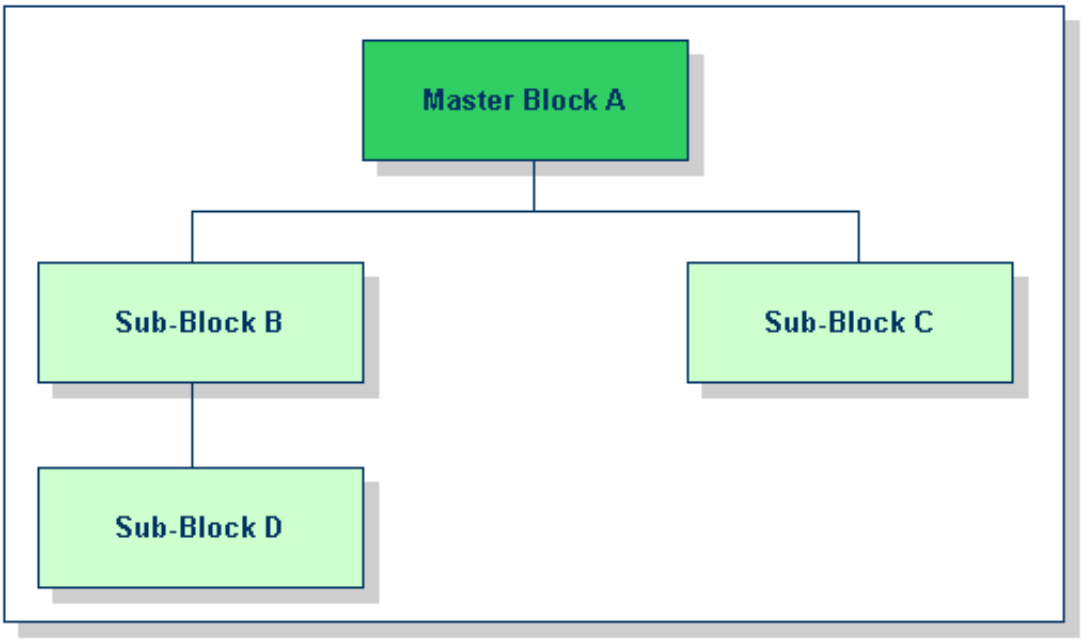
```
DEFINE DATA LOCAL
  01 #A (N5/1:10,1:10)
  01 #B (N5/1:10,1:10)
  01 #C (N5)
  END-DEFINE
  ...
```

1. **ADD #A(*,*) TO #B(*,*)**
   The result operand, array #B, contains the addition, element by element, of the array #A and the original value of array #B.
2. **ADD 4 TO #A(*,2)**
   The second column of the array #A is replaced by its original value plus 4.
3. **ADD 2 TO #A(2,*)**
   The second row of the array #A is replaced by its original value plus 2.
4. **ADD #A(2,*) TO #B(4,*)**
   The value of the second row of array #A is added to the fourth row of array #B.
5. **ADD #A(2,*) TO #B(*,2)**
   This is an illegal operation and will result in a syntax error. Rows may only be added to rows and columns to columns.
6. **ADD #A(2,*) TO #C**
   All values in the second row of the array #A are added to the scalar value #C.
7. **ADD #A(2,5:7) TO #C**
   The fifth, sixth, and seventh column values of the second row of array #A are added to the scalar value #C.

Data Blocks

To save data storage space, you can create a global data area with data blocks. Data blocks can overlay each other during program execution, thereby saving storage space.

For example, given the following hierarchy, blocks B and C would be assigned the same storage area. Thus it would not be possible for blocks B and C to be in use at the same time. Modifying block B would result in destroying the contents of block C.



The following topics are covered below:

- Defining Data Blocks
- Block Hierarchies

Defining Data Blocks

You define data blocks in the data area editor. You establish the block hierarchy by specifying which block is subordinate to which: you do this by entering the name of the "parent" block in the comment field of the block definition.

In the following example, SUB-BLOCKB and SUB-BLOCKC are subordinate to MASTER-BLOCKA; SUB-BLOCKD is subordinate to SUB-BLOCKB.

The maximum number of block levels is 8 (including the master block).

Example:

Global Data Area G-BLOCK:

```
  I T L Name                            F Leng Index/Init/EM/Name/Comment
  - - - --------------------------------- - ---- --------------------------------
    B   MASTER-BLOCKA
      1 MB-DATA01                       A   10
    B   SUB-BLOCKB                             MASTER-BLOCKA
      1 SBB-DATA01                      A   20
    B   SUB-BLOCKC                             MASTER-BLOCKA
      1 SBC-DATA01                      A   40
    B   SUB-BLOCKD                             SUB-BLOCKB
      1 SBD-DATA01                      A   40
```

To make the specific blocks available to a program, you use the following syntax in the DEFINE DATA statement:

Program 1:

```
DEFINE DATA GLOBAL
         USING G-BLOCK
         WITH MASTER-BLOCKA
  END-DEFINE
```

Program 2:

```
DEFINE DATA GLOBAL
         USING G-BLOCK
         WITH MASTER-BLOCKA.SUB-BLOCKB
  END-DEFINE
```

Program 3:

```
DEFINE DATA GLOBAL
         USING G-BLOCK
         WITH MASTER-BLOCKA.SUB-BLOCKC
  END-DEFINE
```

Program 4:

```
DEFINE DATA GLOBAL
         USING G-BLOCK
         WITH MASTER-BLOCKA.SUB-BLOCKB.SUB-BLOCKD
  END-DEFINE
```

With this structure, program 1 can share the data in MASTER-BLOCKA with program 2, program 3 or program 4. However, programs 2 and 3 cannot share the data areas of SUB-BLOCKB and SUB-BLOCKC because these data blocks are defined at the same level of the structure and thus occupy the same storage area.

Block Hierarchies

Care needs to be taken when using data block hierarchies. Let us assume the following scenario with three programs using a data block hierarchy:

Program 1:

```
DEFINE DATA GLOBAL
         USING G-BLOCK
         WITH MASTER-BLOCKA.SUB-BLOCKB
  END-DEFINE
  *
  MOVE 1234 TO SBB-DATA01
  FETCH 'PROGRAM2'
  END
```

Program 2:

```
DEFINE DATA GLOBAL
         USING G-BLOCK
         WITH MASTER-BLOCKA
  END-DEFINE
  *
  FETCH 'PROGRAM3'
  END
```

Program 3:

```
DEFINE DATA GLOBAL
         USING G-BLOCK
         WITH MASTER-BLOCKA.SUB-BLOCKB
  END-DEFINE
  *
  WRITE SBB-DATA01
  END
```

Program 1 uses the global data area G-BLOCK with MASTER-BLOCKA and SUB-BLOCKB. The program modifies a field in SUB-BLOCKB and FETCHes program 2 which specifies only MASTER-BLOCKA in its data definition. Program 2 resets (deletes the contents of) SUB-BLOCKB. The reason is that a program on level 1 (for example, a program called with a FETCH statement) resets any data blocks that are subordinate to the blocks it defines in its own data definition. Program 2 now FETCHes program 3 which is to display the field modified in program 1, but it returns an empty screen. For details on program levels, see Object Types.

